Lab 4 Acceptance testing with web automation (Selenium)

Context and key points

Key Points

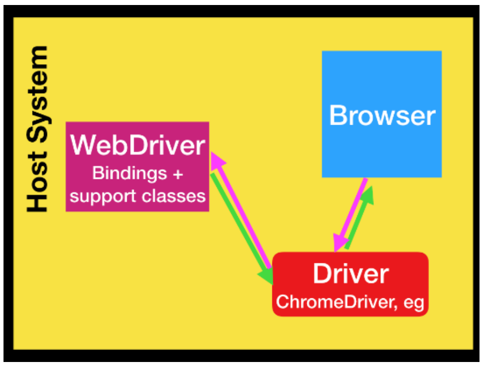
- Acceptance tests (or functional test) exercise the user interface of the system, as if a real user was using the application. The system is treated as a black box.
- Browser automation (control the browser interaction from a script) is an essential step to implement acceptance tests on web applications. There are several frameworks for browser automation (e.g.: Puppeteer); for Java, the most used framework is the WebDriver API, provided by Selenium (that can be used with JUnit engine).
- Selenium is an umbrella project for a range of tools and libraries that enable and support the automation of web browsers.
- The test script can easily get "messy" and hard to read. To improve the code (and its maintainability) we could apply the Page Objects Pattern.
- Web browser automation is also very handy to implement “smoke tests”.

Explore

- Book “Hands-On Selenium WebDriver with Java”
- Another Page Object Model example. Criticism on the Page Object Pattern for modern web apps (and alternatives).

4.1 WebDriver starter

Selenium WebDriver offers a concise programming interface (i.e., API) to drive a (web) browser, as if a real user is operating the browser.



A) Note the “getting started” section available from Selenium documentation (dependencies, instantiating the browser binding, etc…).

Use method #2 (better) or #3 explained in “Install drivers” page. For #3, you need to download the driver implementation for your browser and add driver implementation to the system PATH.

B) Implement the example discussed in the “hello world” section of B. Garcia’s book (Example 2-1).

Main dependencies for POM.xml:

- `org.seleniumhq.selenium:`**`selenium-java`**
- `io.github.bonigarcia:` **`selenium-jupiter`**
- `org.junit.jupiter:`**`junit-jupiter`**

4.2 Selenium IDE recorder

Usually, you can use the Selenium IDE to prepare/record your tests interactively and to explore the “locators” (e.g.: id for a given web element).

A) Record the test interactively
Install the Selenium IDE plug-in/add-on for your browser.

Using the https://blazedemo.com/ dummy travel agency web app, record a test in which you select a and buy a trip. Be sure to add relevant “asserts” or verifications to your test.
Replay the test and confirm the success. Also experiment to break the test (e.g.: by explicitly editing the parameters of some test step).

Add a new step, at the end, to assert that the confirmation page contains the title “BlazeDemo Confirmation”. Enter this assertion “manually” (in the editor, **not** recording).

Be sure to save you Selenium IDE test project (it creates a *.side file, to be included in your git).

B) Export and run the test using the Webdriver API
Export the test from Selenium IDE into a Java test class and include it in the previous project.
Refactor the generated code to be compliant with JUnit 5. [Note: adapt from exercise 1]
Run the test programmatically (as a JUnit 5 test).

C) Refactor to use Selenium extensions for JUnit 5
JUnit 5 allows the use of extensions which may provide annotation for dependency injection (as seen previously for Mockito). This is usually a more compact and convenient approach.

The Selenium-Jupiter extension provides convenient defaults and dependencies resolution to run Selenium tests (WebDriver) on JUnit 5 engine.
Note that this library will ensure several tasks:

- enable dependency injection with respect to the WebDriver implementation (automates the use of WebDriverManager to resolve the specific browser implementation). You do not need to pre-install the WebDriver binaries; they are retrieved on demand.
- if using dependency injection, it will also ensure that the WebDriver is initialized and closed.

Refactor your project to use the Selenium-Jupiter extension (@ExtendWith(SeleniumJupiter.class; use a dependency injection to get a “browser” instance; no explicit “quit”).

4.3 Page Object pattern

Consider the example discussed here (or, for a more in depth discussion, here).
Note: the target web site implementation may have changed from the time the article was written and the example may require some adaptations to run (i.e., pass the tests). However, it is not mandatory to have the example running.
“Page Object model is an **object design pattern** in Selenium, where webpages are represented as classes, and the various elements [of interest] on the page are defined as variables on the class. All possible user interactions [on a page] can then be implemented as methods on the class.”

A) Implement the “Page object” design pattern for a cleaner and more readable test using the same application problem from exercise 4.2. In a new project, refactor the previous implementation to use the Selenium-Jupiter extension and Page Object pattern.

4.4 Browser variations

Consider using a browser that is not installed in your system. You may resort to a Docker image very easily (see Docker browsers section).
Note that, in this case, the WebDriver will connect to a remote browser (no longer direct communication) and you should Docker installed in your system.